Keymap.h - Evaluation

Student #20107104674840

Write a brief (1-2 pages) evaluation report of your map class: justify any design decisions (e.g. its interface, the use of friends) and evaluate the support provided by C++ for developing abstract data types and the use of this support for your map class.
Identify any weaknesses in your implementation and discuss improvements and alternative representations, and compare your map class with the map class provided in the STL

Design Decisions

My map was implemented as a single class which maps pointers of `K:V` values together in two arrays.
C++ has native support for abstract data types, which allowed me to introduce the `K:V` type-mappings with relative ease after desiging a map compatible with only a single datatype, however it also requires that all type definitions and code be kept in a header file, rather than the `.cpp` files.
I have used the generic type mapping of `K` and `V` to represent Key and Value pairs, and the map should operate with any type `V`, and any type `K`, including custom classes which have a valid `==` operator, as long as they aren't `nullptr` values.
Some maps dissallow `NULL`, however I opted to not dissallow "null" values, as the C++ keyword `NULL` compiles to `0`. However the implementation will not accept insertion of `nullptr`s into the map.

The map is iterable both forward and backward, and does so by implementing a `friend` class called `MapIter` - this allows the iterator to access the private and protected members of the `Keymap`, without having to request it through any `getter` methods. Due to the circular relationship between the iterator and the map, the `MapIter` is forward-declared in order to access the `Keymap`, and stores pointers to the map. The map itself also stores pointers to it's iterator.

The specific implementation of Keymap produced in this assignment utilises two parallel arrays of pointers in order to store data; that is to say, the map doesn't directly store data, it simply stores the locations to the data. This should make manipulations of the map, updating, removing, etc, relatively efficient due to the small size of memory pointers, when compared to the data they point toward. This decision was made to reduce duplication of objects in memory, in theory leaving a smaller footprint. Whenever temporary arrays are no longer needed, they are purged from memory using the `delete` keyword.
Similarly, some keymaps wrap the `key:value` pair, and then store the wrapped pair in a single array. I opted against this to try and make accessing the data within the map simpler and more direct.

Insertion is linear and removal preserves order; insertion as a result is fast, as it simply checks the current `length` of the array, and inserts the pair at `array[length+1]`. Removal requires finding the element, and copying everything from the right-hand-side of the element one location to the left, which can be especially slow on larger datasets.

Improvements

In terms of usability, I have implemented many of my methods with names that do not align completely with other standards, such as the STL library, which may be confusing to some. To improve this would be a simple case of refactoring which could be done easily at a later date.

While the insertion is fairly lightweight, my delete method could be improved significantly. The current implementation iterates over the entire array, and copies elements to a new array while skipping over the element to delete. This operates in the order of `O(n)`, which works fine for smaller datasets however will become slower over larger collections. This could be reduced to `O(log(n))` if I implemented a binary search instead of simply iterating over every element, however depends on the key values being sort-able and that the insertion methods enforces sorting.

If speed of deletion is considered more important than preservation of insertion order, then I could omit the copying stage of the delete function, and make the insert method check for the first available location storing no value, or a `nullptr`.

There are also several methods available in the C++ STL library which were not implemented in my custom map class.

The final issue I have with my code is that it is all held in a single class, and some of it could be abstracted away - such as the array size management, and - which would offer two primary benefits; it would enhance the readability by moving unecessary features to a different codebase, and make it more maintainable as a result.